

CRC-Cards and Roleplay Diagrams—Informal Tools to Teach OO Thinking

Jürgen Börstler

Umeå University, Sweden
 jubo@cs.umu.se

1 Introduction

CRC-cards (**C**lass, **R**esponsibility and **C**ollaborator) are a lightweight approach to collaborative object-oriented modelling that has been developed as a tool for teaching object-oriented thinking to programmers [1]. They have been used widely in various teaching and training contexts [2, 6, 7, 10].

A CRC-card (see figure 1) corresponds to a **class**.

A **responsibility** is something the objects of a class know or do as a service for other objects. A book object in a library application, for example, might, among other things, be responsible for checking itself out and knowing its title and due date (see *Book* in 1). The responsibilities of the objects of a class are written along the left side of the card.

A **collaborator** is an object of another class “helping” to fulfil a specific responsibility. A book object, for example, can only know whether it is overdue, if it also knows the current date. In figure 1 this information is provided by a collaborator object of class *Date*. The class names of collaborators are written along the right side of the card.

The back of the card can be used for a brief description of the class’ purpose, comments and miscellaneous details.

Class: <i>Book</i>		Class: <i>Librarian</i>	
Responsibilities	Collaborators	Responsibilities	Collaborators
<i>knows whether on loan</i>		<i>check in book</i>	<i>Book</i>
<i>knows due date</i>		<i>check out book</i>	<i>Book, Borrower</i>
<i>knows its title</i>		<i>search for book</i>	<i>Book</i>
<i>knows its author(s)</i>		<i>knows all books</i>	
<i>knows its registration code</i>		<i>search for borrower</i>	<i>Borrower</i>
<i>knows if late</i>	<i>Date</i>	<i>knows all borrowers</i>	
<i>check out</i>			
Class: <i>Borrower</i>		Class: <i>Date</i>	
Responsibilities	Collaborators	Responsibilities	Collaborators
<i>knows its name</i>		<i>knows current date</i>	
<i>keeps track of borrowed items</i>		<i>can compare two dates</i>	
<i>keeps track of overdue fines</i>		<i>can compute new dates</i>	

Fig. 1. A set of CRC-cards for modelling a simple library application.

2 Using CRC-Cards

The CRC-card approach can be roughly divided into two stages; (1) development of an initial CRC-card model and (2) scenario definition and roleplaying. In the roleplaying, participants “act-out” predefined scenarios, much like actors following a script when playing the characters in play. In object-oriented development, the characters of the play are the objects in a software system and the scenarios are hypothetical but concrete situations of system usage. The roleplaying helps students “learn to think in objects” in the following way [1]:

1. Students can only control their own assigned role(s) (i.e. CRC-card(s)); this helps them to give up global control [8].
2. Students can only act according to the predefined responsibilities of their role(s); this helps them to focus on design, i.e. distribution of responsibilities [10].
3. Students collaboratively enact how their system should work; this immerses them in the “object-ness” of the material [9].

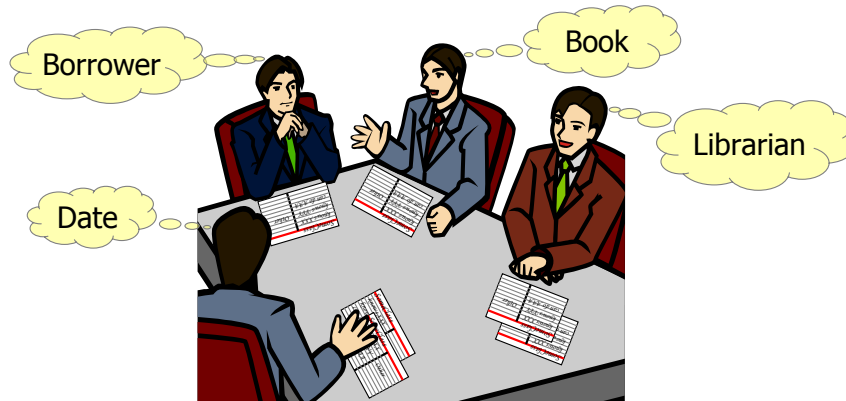


Fig. 2. Roleplaying is a collaborative activity.

We have successfully used CRC-cards in our introductory programming courses for many years. It forces participants to reason about models and explain design decisions to their peers. This makes the approach particularly useful for collaborative active learning. However, we have also noticed a few problems, in particular with respect to the roleplaying (see [5] for more details):

1. *CRC-cards blur the difference between objects and classes.* CRC-cards serve two “roles”; they are used as surrogates for classes in model development and as surrogates for objects in the roleplaying. This makes it even more difficult for the students to understand the important differences between class and object.
2. *It is unclear what makes a “good” scenario.* Ambiguous or unclear scenarios leave too many degrees of freedom and make exploration very difficult.
3. *Students get easily “lost” during the roleplaying.* Roleplaying is explorative, i.e. many different paths are “tested” and decisions about the kind and order of messages send are revised and refined continuously. This makes it very difficult to backtrack to a consistent state in a scenario, in particular, when the underlying design is changed (which might make earlier decision invalid).
4. *There are no guidelines for the mechanics of the roleplaying activities.*
5. *There are no guidelines for the documentation of scenarios and roleplays.*

3 Roleplay Diagrams

To support the roleplaying, we needed a to help the students keeping track of their decisions. A simple and intuitive notation should make it possible to successively built up the documentation of the roleplay “on the fly” as the scenario unfolds. This documentation must furthermore be very easy to change. Otherwise, the dynamics of the roleplaying would be disturbed. The documentation should support the roleplaying, not the other way around.

These requirements ruled out raw text or existing UML diagrams [4]. We tried raw text and sequence diagrams in the beginning, but they are too difficult to change and reorganize. Another disadvantage of sequence diagrams is their lack of structural information. Structural information is, however, very important for capturing all relevant status information at scenario start to make clear which objects are allowed to communicate. Furthermore, it is

impossible to describe attribute values in sequence diagrams (as well as in collaboration or communication diagrams). The only UML diagram covering structural information and attributes is the object diagram. They, however, do not capture message passing, the most important aspect of scenario documentation.

We therefore developed roleplay diagrams (RPDs) by combining elements from UML's object and collaboration diagrams ¹. Objects in RPDs are represented by object "cards". An object card is an instance of a CRC-card showing an object's name, its class and the properties relevant for the current scenario. In figure 3, for example, we have an object `book1`, which is an instance of our CRC-card (class) `Book` from figure 1. For each new object, we create a new object card and put it on the white-board or a large sheet of paper. This helps to make clear the distinction between objects and their corresponding classes.

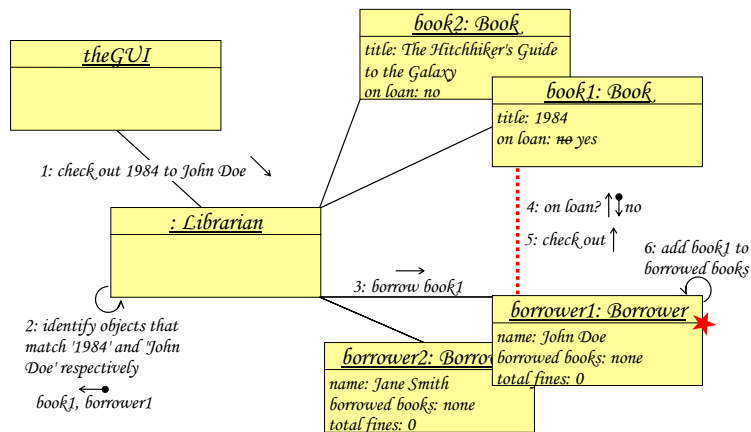


Fig. 3. Snapshot of an evolving RPD for checking out book "1984" to borrower "John Doe".

Object to object communication is only possible if the participating objects "know" each other in the particular scenario. Objects that know each other are connected by a line. Please note, that two objects can only be connected when they are collaborators (see CRC-cards). A service requested (message sent) by an object must correspond to a responsibility listed on the CRC-card corresponding to the serving object. In our example, `on loan?` and `check out` correspond to `Book`'s responsibilities `knows whether on loan` and `check out`, respectively (see figure 1). If there are no corresponding responsibilities, as for example for message 3 in our example, the design (i.e. CRC-card model) must be revised. Messages are documented on the connecting lines between the communicating objects. Small arrows denote the direction of messages. A simple numbering scheme is used to keep track of the ordering of messages. Messages can be annotated with information that is returned.

When the "knowledge" of an object changes, the corresponding object card is updated accordingly. After checking out `book1` its property `on loan` is changed to `yes`. At the end of a scenario roleplay, the RPD documents exactly what happened, including relevant state changes.

Please note that the connection between `book1` and `borrower1` (dotted line in figure 3) is not present at the start of the scenario. It is established as a result of sending message 3.

4 Experience

Our experience with CRC-cards and RPDs is very positive [5, 6]. The approach supports "object thinking" and helps students understand the dynamics of object-oriented programs.

¹ Actually, RPDs are closer to Booch's original object diagrams [3] than any of the current UML diagrams

The RPDs are useful for defining the start situation of a scenario and then recording the roleplaying as the scenario evolves. This makes RPDs an excellent tool for scenario tracking, testing, and documentation. Design decisions can be “tested” without writing a line of code. Some of our students actually use the CRC/RPD approach as their design “method” of choice in other courses.

In table 1, we summarize results from the course evaluations of all offerings of our introductory programming course for the period spring 2001–spring 2005. In these evaluations students can report problems in free form, which we divided into

We have grouped the 12 offerings into four groups; courses covering neither CRC-cards or RPDs, but including GUIs (column `no CRC, no RPD`, 2 offerings); courses covering “pure” CRC-cards (column `CRC`, 6 offerings); and courses using the proposed CRC/RPD approach (column `CRC/RPD`, 3 offerings). One offering, covering GUIs and CRC-cards was excluded from the summary, since it did not fit a category. Please refer to [5, 6] for more details.

Table 1. Percentage of students reporting problems in course evaluations, by problem category.

	no CRC, no RPD	CRC	CRC/RPD	avg. CRC & CRC/RPD
Students answering, total	126	156	111	267
Students reporting problems	55.61	61.91	38.87	54.23
Problem categories:				
- OO thinking	3.86	3.38	3.18	3.31
- OO concepts	10.38	4.17	2.77	3.71
- Design/CRC/RPD	1.59	4.02	7.95	5.33
- Coding assignments	8.56	14.68	1.82	10.39
- “Everything”	6.44	9.74	10.84	10.11
- I/O	4.17	10.29	3.81	8.13
- GUI	12.05	0	0	0
- Other	8.56	15.63	8.50	13.25

Although the data in table 1 is difficult to interpret, it seems to support our claim that CRC/RPD has a positive effect on the understanding of object-oriented concepts. These results are particularly remarkable, since student cohorts in the beginning of the period reported here have been generally stronger than those towards the end of the period.

We specifically want to stress that the CRC/RPD approach does not require any prerequisite knowledge in programming. In fact, students with good programming knowledge often have difficulties focusing on responsibilities, which are on the analysis or design level. They often start discussions about irrelevant low level details. Since their arguments can be very convincing for inexperienced programmers, we strongly recommend to carefully monitor the discussions and interrupt such discussions. Building inhomogeneous teams helps to avoid such problems.

Our experience shows that roleplaying is difficult. The introduction of RPDs helped to overcome most problems. To give students a good start, we have developed a tutorial with guidelines and examples. Furthermore, we present a “live-demo” in front of the class.

In general, students are positive toward the CRC/RPD approach. When asked “Have the CRC exercises been rewarding?”, on average about 2.5 times as many students are clearly positive than clearly negative.

To make using the CRC/RPD approach successful, we strongly recommend the following:

Carefully distinguish between classes and objects. Avoid using CRC-cards as substitutes for objects during the role-play.

Make scenarios as specific as possible. Restrict the degrees of freedom to a few things that should be explored. One can easily get “lost” in long or open-ended scenarios. A good scenario should have a clear goal, well-defined preconditions and an expected outcome.

Start with the simplest possible meaningful scenario. It is important to make the first role-play a successful activity to motivate students to use the technique.

Initialize the role-play properly. Carefully set up all objects that already exist at scenario start (for collections choose a few representative elements). Initialize objects with all relevant information as stated in the preconditions for the scenario. Introduce a single GUI object that handles all communication with users and external systems and connect it to at least one object in the model.

Be careful with object names. Novices easily confuse objects, properties, and references, when they are named similarly.

References

1. Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA'89*, pages 1–6, 1989.
2. Robert Biddle, James Noble, and Ewan Tempero. Reflections on CRC cards and OO design. In *Proceedings Tools Pacific 2002*, pages 201–205, 2002.
3. Grady Booch. *Object-Oriented Analysis and Design with Applications, 2nd edition*. Addison-Wesley, Reading, MA, 1994.
4. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.
5. Jürgen Börstler. Improving CRC-card role-play with role-play diagrams. In *OOPSLA'05 Addendum to the Proceedings (Educators' Symposium)*, pages 356–364, 2005.
6. Jürgen Börstler, Thomas Johansson, and Marie Nordström. Introducing oo concepts with CRC cards and Bluej-a case study. In *Proceedings FIE'02*, pages T2G-1–T2G-6, 2002.
7. Jim Coplien. Experience with CRC cards in AT&T. *C++ Report*, 3(8):1,4–6, 1991.
8. Mark Guzdial. Centralized mindset: A student problem with object-oriented programmings. In *Proceedings SIGCSE'95*, pages 182–185, 1995.
9. David West. *Object Thinking*. Microsoft Press, Redmond, WA, 2004.
10. Nancy Wilkinson. *Using CRC Cards, An Informal Approach to Object-Oriented Development*. SIGS, New York, NY, 1995.